1. 上课约定须知
2. 上次作业复盘
3. 上次内容总结
4. 本次内容大纲
5. 详细课堂内容
    5. 1. MapReduce架构设计
        5. 1. 1. MapReduce 逻辑设计
        5. 1. 2. MapReduce 物理设计
    5. 2. MapReduce的核心Shuffle流程
    5. 3. MapReduce程序编写规范总结
    5. 4. MapReduce并行度决定机制
        5. 4. 1. MapTask并行度决定机制
        5. 4. 2. ReduceTask并行度决定机制
    5. 5. MapReduce序列化和分区分组
        5. 5. 1. 关于 MapReduce 的自定义分区
        5. 5. 2. 关于 MapReduce 的序列化
        5. 5. 3. 关于 MapReduce 的自定义分组
    5. 6. MapReduce Join实现
        5. 6. 1. MapJoin 实现
        5. 6. 2. ReduceJoin 实现
6. 本次课程总结
7. 本次课程作业

1. 上课约定须知

课程主题：MapReduce--第一次课（架构设计和企业最佳实践）
上课时间：20:00 - 23:00
课件休息：21:30 左右 休息10分钟
课前签到：如果能听见音乐，能看到画面，请在直播间扣 666 签到

2. 上次作业复盘

1、ZooKeeper作业：实现一个类似于 ZooKeeper 的数据模型存储系统。具备基本的树形结构和增删改查功能，并且能进行冷启动数据恢复

2、HBase作业：实现一个类似于 HBase 的海量 key-value 存储系统。具备基本的低延时的随机读写功能，并且能进行冷启动恢复。

3. 上次内容总结

到此为止：我们已经学完了 ZooKeeper 和 HBase，总的来说，7次课，分别简单总结一下：

第一个知识点：ZooKeeper

```
1、ZooKeeper 相关的各种分布式理论
    2PC  3PC  Quorum机制  NWR读写模型  PAXOS/RAFT/ZAB  CAP/BASE理论
2、ZooKeeper 企业最佳实践
    ZK数据模型系统+监听机制  ZK应用场景和设计目的  ZK企业最佳实践：分布式锁，选举等
3、ZooKeeper 源码分析01
    zookeeper基础功能支撑  zookeeper集群启动  zk选举准备（startLeaderElection()）
4、ZooKeeper 源码分析02
    zookeeper选举（lookForLeader()）  zookeeper状态同步
```

第二个知识点：HBase

```
1、HBase的设计思想头脑风暴
    LSM-Tree + 跳表结构  HBase架构设计  HBsse核心概念  HBase表模型
2、HBase源码分析01
    HBase集群启动
3、HBase源码分析02
    DDL创建表  DML插入数据  flush动作
```

第三个知识点：到此为止的四大技术点总结

四个知识点：三大块（存储块，计算块，其他）

```
1、HDFS          分布式文件系统
2、Kafka         分布式消息系统
3、ZooKeeper     分布式协调服务组件
4、HBase         分布式key-value类型的NoSQL数据库
```

4. 本次内容大纲

从今天开始，给各位讲解两次的 MapReduce，了解最开始的分布式计算引擎是如何一回事。

请注意：计算引擎分为两部分：编写分布式应用程序的API + 资源管理调度引擎，新版本的 MapReduce 就只是一套编程 API，最终帮助程序调度资源运行起来的是 YARN。同样，Storm，Spark，Flink 等也都是包含这两部分，并且也都可以运行在 YARN 当中。

今天是 MapReduce（编程模型 一套编写分布式应用程序的API） 的第一次课，主要讲解 MapReduce 的核心运行机制 和 企业最佳实践。

```
1、MapReduce 的架构设计和工作原理
    MapReduce架构设计
    MapReduce的核心Shuffle流程
    MapReduce并行度决定机制
2、MapReduce 企业最佳实践
    MapReduce程序编写规范总结
    MapReduce序列化和分区分组
    MapReduce Join实现
```

5. 详细课堂内容

5.1. MapReduce架构设计

两个问题：

```
1、真实的产生背景：google最早研发并发布论文
2、一个简单的问题：一个书架（我：一个人） + 一个图书馆（馆长：一个管理团队）
```

既然复杂问题，单台计算搞不定，那么就发挥人多力量的优势：组建一个多服务器组成的集群来搞定分布式并行计算。核心过程为：

```
1、第一阶段 Mapper（提取特征的过程）：复杂大任务拆分成多个小任务并行执行计算
2、第二阶段 Reducer（执行逻辑的地方）：把第一阶段的并行执行的小任务的执行结果进行汇总
```

MapReduce：所以一句话总结：**分而治之 + 并行计算。**

把单机计算程序，扩展成分布式计算应用程序，会遇到非常多的问题：

```
1、数据存储的问题，首先需要搞定海量数据存储的问题。
2、运算逻辑至少要分为两个阶段，先并行计算（map），然后汇总（reduce）结果
3、这两个阶段的计算如何启动？如何协调？
4、运算程序到底怎么执行？数据找程序还是程序找数据？
5、如何分配两个阶段的多个运算任务？
6、如何管理任务的执行过程中间状态，如何容错？
7、如何监控和跟踪任务的执行？
8、出错如何处理？抛异常？重试？
```

计算框架：具备的能力：阶段的控制，任务的协调，跟踪，监控，容错等各种通用功能！

真正的计算逻辑是不能封装的！定义好处理流水线！通用的部分给默认实现，需要用户自定义的地方，定义规范。

一个分布式计算应用程序的执行会分成很多个步骤：

```
读取数据 + 执行计算1 + 数据混洗阶段 + 执行计算2 + 输出结果

具体的MapReduce实现；
读取数据：InputFormat + RecordReader，HDFS（TextIntputFormat + LineRecordReader）
执行计算1：Mapper（规范：逐行处理，接收一条数据，返回一条数据 ）
          提取待计算数据value的特征key
数据混洗阶段：Shuffle（决定Mapper阶段的数据如何被分发到Reducer的逻辑）
执行计算2：Reducer（规范：聚合分析，聚合处理：接收一组数据，返回一条数据）
         经过上一个步骤的数据分发，现在当前节点已经接受到某个特征的所有的待计算的值，然后执行汇总逻辑即可
输出结果：OutputFormat + RecordWriter，HDFS（TextOutputFormat + LineRecordWriter）
```

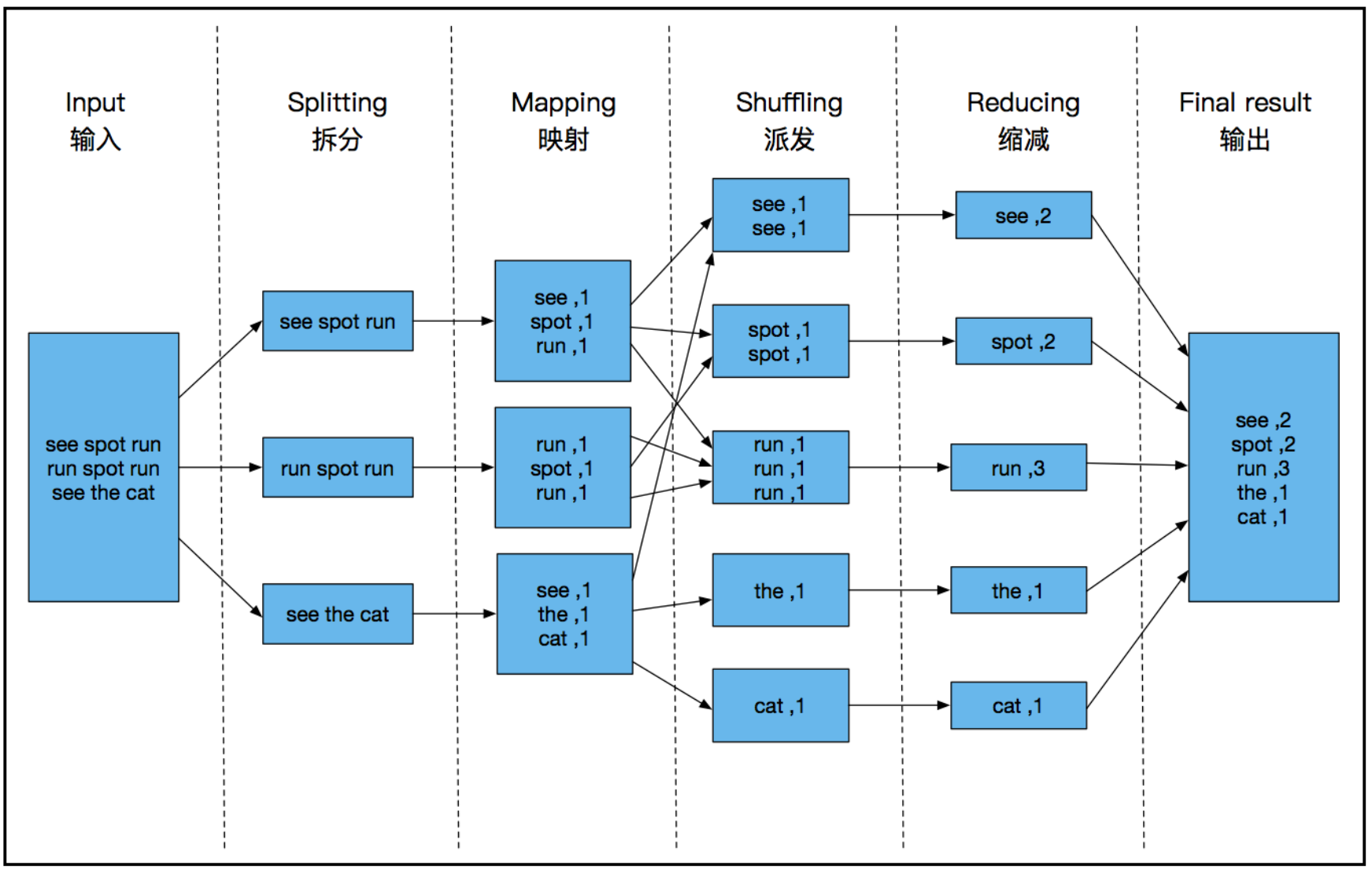
MapReduce: 计算框架 = 通用功能的封装 + 自定义的处理逻辑

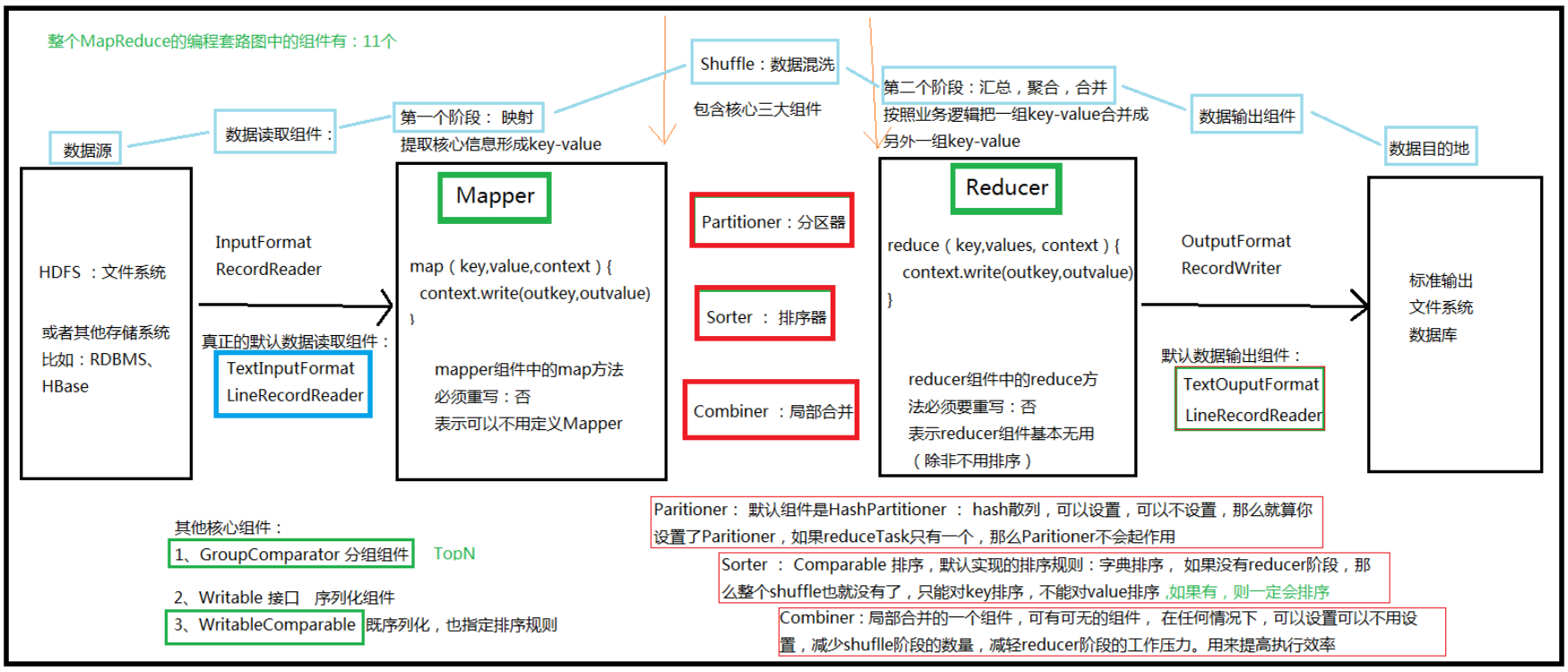
5.1.1. MapReduce 逻辑设计

首先来看**逻辑实现**：以任何生活场景为例，当一件事变得复杂了之后，原来由一个人可以干完的活儿，就需要由一个团队去做，这就涉及到怎么切分任务，怎么调度跟踪监控任务的执行。

WordCount 单词计数（单词切割的时候，得到是一个个的单词，到底应该把那些个单词对应的一次汇总成最终的次数呢？）最大的问题：一定要把所有的该加起来合并的1一定要收集到一起来执行汇总

```
数据源 ---->
    数据源：hdfs hbase MySQL 数据读取组件
    设计通用实现：InputFormat + RecordReader
    默认实现：TextInputFormat + LineRecordReader

第一阶段的逻辑计算 ----> 其实就是给每一个需要计算的value打上标记（提取特征）
    可能的计算逻辑：复杂多样
    设计通用实现：定义好怎么给数据执行处理，怎么收集结果，到底如何计算流程用户自定义
    默认实现：MapTask，核心：执行 mapper.run()(setUp() + map() + cleanup())
    （其实就是针对原始数据进行改造，提取需要计算的数据value，并加上特征key）
    真正的有意义的理解：把待计算的value提取到一个key，用来作为标识，以方便，在网络传输过程中，知道哪些value应该被传送到那些第二阶段的计算节点上!

两阶段之间的中间过程 ---->
    数据传输可能性：需要根据第一阶段提取的key，来进行value的汇总
    设计通用实现：按照待计算的数据value的特征key进行网络传输，把相同的key的value传输到同一个节点作
    默认实现：Shuffle机制（Partitioner Sorter Combiner）

第二阶段的逻辑计算 ----> 通过上一个步骤的网络数据传送，相同特征的所有的value已经被汇总到当前节点
    可能的计算逻辑：复杂多样
    设计通用实现：定义好怎么给数据执行处理，怎么收集结果，到底如何计算流程用户自定义
    默认实现：ReduceTask，核心：执行 reducer.run()(setUp() + reduce() + cleanup())
    （执行之前，框架已经帮助我们完成了关于相同特征数据收集到一起的问题）

数据目的地 --->
    数据目的地：hdfs hbase MySQL
    设计通用实现：专门负责数据写出的组件：OutputFormat + RecordWriter
    默认实现：TextOutputFormat + LineRecordWriter
```





5.1.2. MapReduce 物理设计

再来看**物理实现**：任何一个分布式应用程序，首先都必须具有一个主控程序，我们姑且称之为 AppMaster，由于 MapReduce 会按照两阶段执行的思路来做，所以如何在调度完第一阶段的 Task 执行结束之后，再去调度第二阶段 Task 的执行是重中之重。

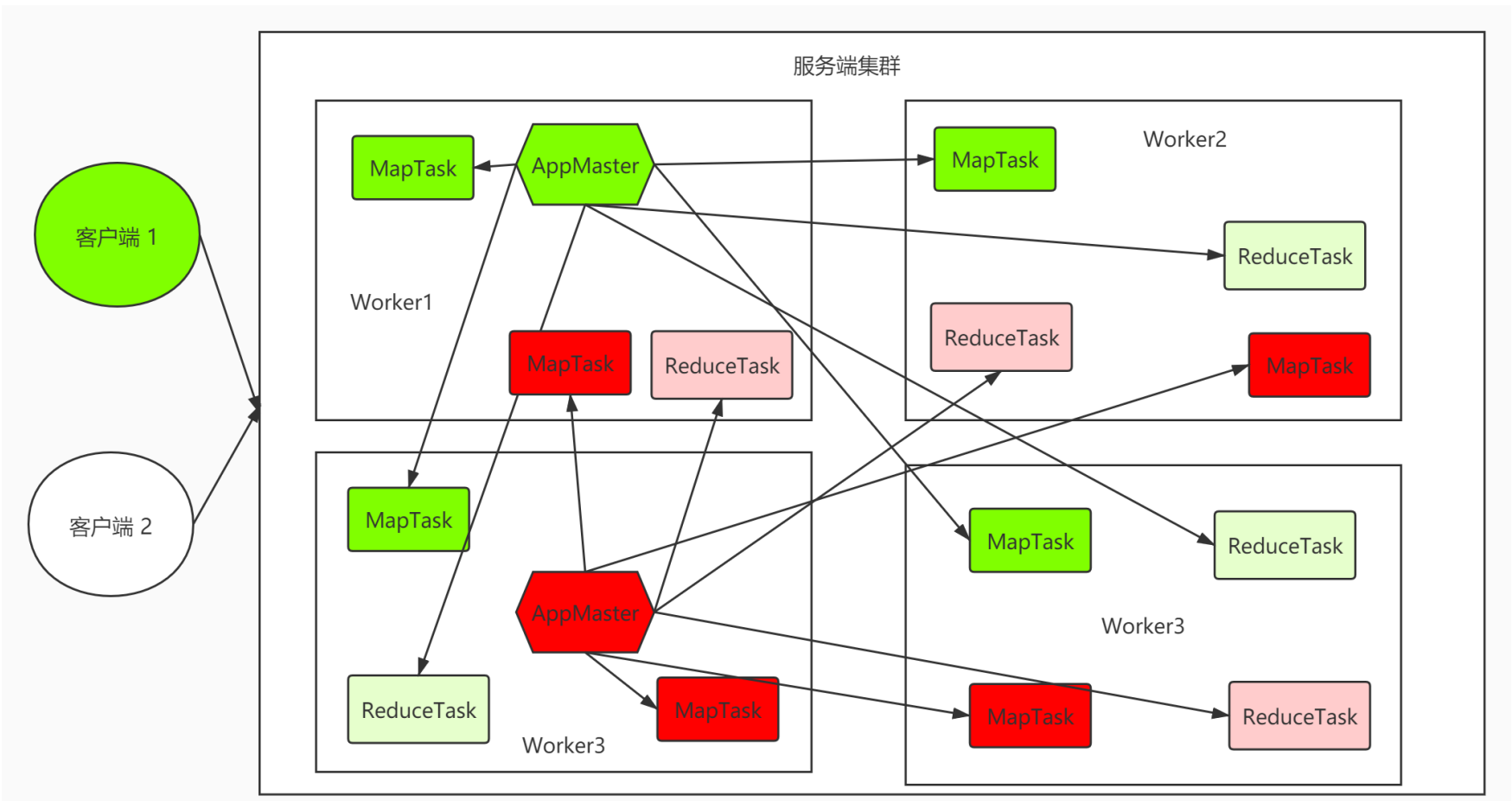
完整的一个分布式应用程序的执行：

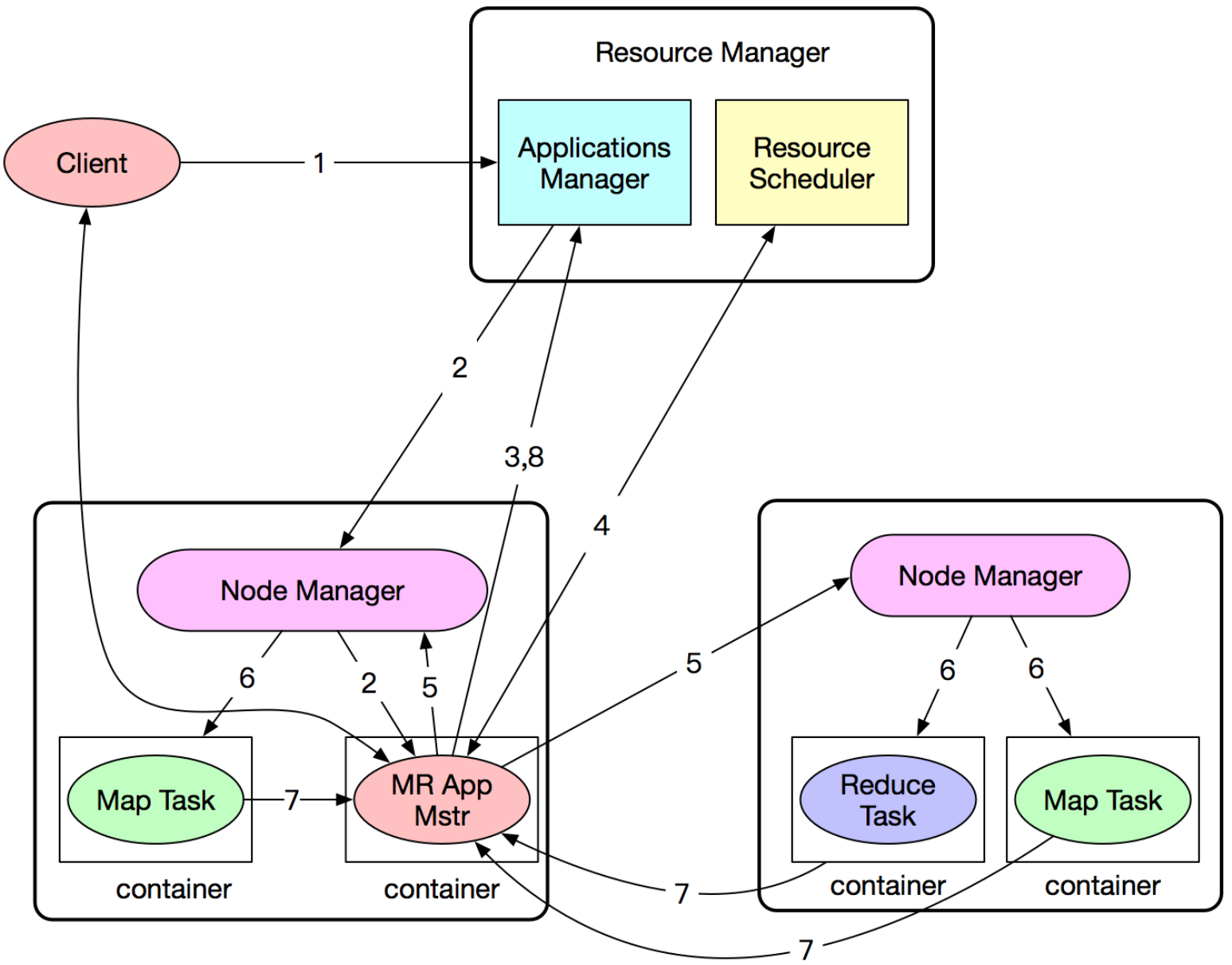
```
1、第一阶段：N个MapTask
2、第二阶段：N个ReduceTask
```

```
分布式集群     多个分布式计算应用程序     AppMaster     Task
一家公司  老板        多个项目团队          Leader      成员
```

为什么在 Hadoop -2.x 版本中，把原来的 MapReduce 一分为二，变成 MapReduce（编程框架） 和 YARN（资源调度） 两个组件？
Spark（Application Job Stage Task）

整体思路如下：





分布计算的核心思想：多阶段的多Task并行计算框架

5.2. MapReduce的核心Shuffle流程

关于 MapReduce 的复杂性的解释：

```
1、从功能上：从第一个阶段的每个Task执行节点中，把具有相同特征key的数据value分发到同一个节点。每个节点都得做这个事
2、从过程上：
    第一个阶段的所有 Task 在理论上来说，都需要给第二个阶段的每个 Task 传送数据
    第二个阶段的所有 Task 在理论上来说，都需要从第一个阶段的每个 Task 拉取数据
3、从效率上：
    最终第二个阶段的任意一个Task都会拉取到上游阶段的很多个结果文件
    假设现在想要获取 第一个特征所对应的所有的value：把所有文件都扫描一次，拿出来当前这个key对应的所有value，然后来执行一次 reduce 方法的逻辑
    如果需要获取第二个key的所有数据，那么还是使用同样的方式!
    假如现在这个 ReduceTask 有一万个不同的key，那么这些文件需要被扫描一万次
```

问题是什么?

```
现在每拿取到一组数据，就需要从所有数据文件中，扫描一次！简化一下：就是一个乱序文件！
```

解决方案是什么？针对所有的拉取到的数据文件，执行合并排序！效果：扫描文件一次，每扫描一段，就得到一组数据

```
hello,1
world,1
hello,1
world,1
```

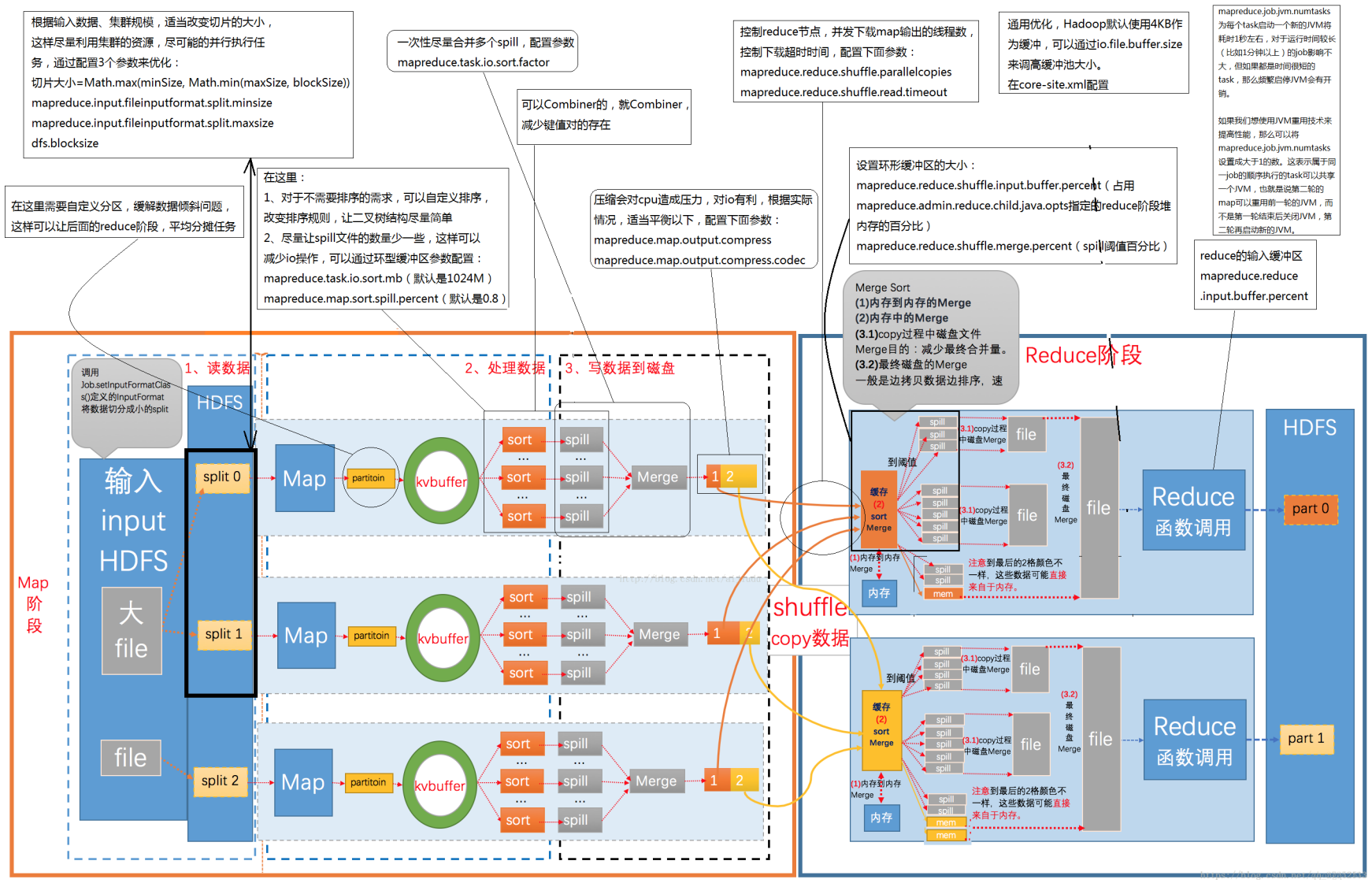
```
hello,1
hello,1
world,1
world,1
```

```
父辈：四兄弟
子辈：六兄弟
```

```
1、为什么 MapReduce 有 Reducer 阶段，就一定会有 排序的动作？
2、为什么 MapReduce 的数据处理模型是 Key-Value 类型？
```

Shuffle 是 MapReduce 应用程序执行过程中，最重要的一个过程。也是所有的分布式计算引擎，都必须支持的一个工作阶段。

详细的内容：请参考我的博客：https://blog.csdn.net/zhongqi2513/article/details/78321664



5.3. MapReduce程序编写规范总结

MapReduce 的 API 设计类似于 责任链设计模式 的实现！

MapReduce 为什么叫 MapReduce，核心思想来源于一个函数式编程语言：lisp，它是一种函数式的编程语言，里头提供和实现了 map 和 reduce 两种函数。

```
Job（InputFormat + RecordReader Mapper Partitioner Sorter Combiner Reducer OutputFormat + RecordWriter ）

Configuration conf = new Configuration();
Job job = Job.getInstance(conf);

job.setMapperClass(xxx);
job.setReducerClass(xxx);
job.setPartitionerClass(xxxx);
...

job.submit();
```

```java
public class WordCount {

    public static void main(String[] args) throws Exception {
        Job job = Job.getInstance();
        job.setMapperClass(WCMapper.class);
        job.setReducerClass(WCReducer.class);
        job.submit();
    }
    private static class WCMapper extends Mapper<LongWritable, Text, Text, LongWritable>{
        @Override
        protected void map(LongWritable key, Text value,
                Mapper<LongWritable, Text, Text, LongWritable>.Context context)
                throws IOException, InterruptedException {
            // 在此写maptask的业务代码
        }
    }
    private static class WCReducer extends Reducer<Text, LongWritable, Text, LongWritable>{
        @Override
        protected void reduce(Text arg0, Iterable<LongWritable> arg1,
                Reducer<Text, LongWritable, Text, LongWritable>.Context arg2)
                throws IOException, InterruptedException {
            // 在此写reducetask的业务代码
        }
    }
}
```

mapreduce的主入口，其中用job来管理该程序

WordCount的maptask业务代码

WordCount的reducetask业务代码

MapReduce程序编写规范

1、用户编写的程序分成三个部分：Mapper，Reducer，Driver(提交运行MapReduce程序的客户端)
2、Mapper的输入数据是KV对的形式（KV的类型可自定义）
3、Mapper的输出数据是KV对的形式（KV的类型可自定义）
4、Mapper中的业务逻辑写在map()方法中
5、map()方法（mapTask进程）对每一个输入<K,V>调用一次
6、Reducer的输入数据类型对应Mapper的输出数据类型，也是KV对的形式，简单说，mapTask的输出KV就是reduce的输入KV
7、Reducer的业务逻辑写在reduce()方法中
8、ReduceTask进程对每一组相同k的<K,V>组调用一次reduce()方法
9、用户自定义的Mapper和Reducer都要继承各自的父类
10、整个程序需要一个Drvier来进行提交，提交的是一个描述了各种必要信息的Job对象

5.4. MapReduce并行度决定机制

5.4.1. MapTask并行度决定机制

一个完整的 MapReduce 程序在分布式运行时有两类实例进程：

1、MRAppMaster：负责整个程序的过程调度及状态协调　　主控程序
2、Yarnchild：负责map阶段的整个数据处理流程
3、Yarnchild：负责reduce阶段的整个数据处理流程　　第一阶段 第二阶段的 Task

以上两个阶段 MapTask 和 ReduceTask 的进程都是 YarnChild，并不是说这 MapTask 和 ReduceTask 就跑在同一个 YarnChild 进行里

FileInputFormat中默认的切片机制

代码实现：

```java
# 决定核心切片逻辑
List<InputSplit> splits = FileInputFormat.getSplits(JobContext);

    # 计算逻辑切片大小：默认等于 128M = 数据块大小
    long splitSize = computeSplitSize(blockSize, minSize, maxSize)
    # 翻译一下就是求这三个值的中间值，片主要由这几个值来运算决定：
    # blocksize：默认是128M，可通过dfs.blocksize修改
    # minSize：默认是1，可通过mapreduce.input.fileinputformat.split.minsize配置
    # maxsize：默认是Long.MaxValue，可通过mapreduce.input.fileinputformat.split.maxsize配置

    # 负责封装和生成切片对象
    makeSplit(file, start, length, hosts, inMemoryHosts)
```

两个问题：

```
1、200个200M文件执行MapReduce最终启动多少个Task？ 400 个
   200个130M文件执行MapReduce最终启动多少个Task？  200 个

2、关于 mapreduce 执行过程中，跨行处理问题
    第一个 InputSplit，读取数据到下一个 InputSplit 的第一个行分隔符处
    从第二个 InputSplit 开始的每一个InputSplit 都要从第一个行分隔符处开始读取，因为前半行断行，已经被上一个 InputSplit 读取过了。
    最后一个 InputSplit 读取到自然结束即可
```

关注一个细节：在 FileInputFormat 中有一个方法叫做：

```
protected boolean isSplitable(FileSystem fs, Path filename) {
    return true;
}
```

他的作用是用来指定：一个文件在进行处理的时候，是否要分开处理。如果你的需求需要将每个文件单独处理的话，请让这个方法返回：false

5.4.2. ReduceTask并行度决定机制

关于 ReduceTask 的并行度决定机制，很暴力的：

```
1、如果没有自定义数据分区规则，则直接使用如下代码随意设置：
    job.setNumReduceTasks(numTasks);

2、如果有自定义数据分区规则的话，依然通过上述代码进行设置，只不过需要注意的时候，这个 numTasks 就不能随意设置了。需要和自定义分区器的分区数量匹配。否则会出现一些问题：
    如果你设置的分区数 大于 应该设置的标准值：出现了一些空跑 Task，并且输出空结果文件
    如果你设置的分区数 等于 应该设置的标准值：Bingo!!!
    如果你设置的分区数 小于 应该设置的标准值：程序运行报错
```

```
分区器的分区个数    reduceTask的个数   最终reducer阶段的结果文件个数    numTasks
```

如果设置为 0 ， 请注意：

```
不管你 Reducer 定义了什么逻辑，不管你定义了什么 Partitioner 都将失去作用。
最终的 MapReduce 程序只会运行：Mapper 阶段的 Task，然后直接结束。
```

5.5. MapReduce序列化和分区分组

5.5.1. 关于 MapReduce 的自定义分区

默认实现：HashPartitioner

```
public class HashPartitioner<K, V> extends Partitioner<K, V> {
    // 返回值取值范围：[0, numReduceTasks)
    public int getPartition(K key, V value, int numReduceTasks) {
        return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
    }
}
```

如何自定义：

```
class MyPartitioner extends Partitioner{

    int partition(key, value, numberReduceTasks){

        if(phone.substring(0, 3).equals("133")){
            return 0;
        }else if(phone.substring(0, 3).equals("134")){
            return 1;
        }else if(phone.substring(0, 3).equals("135")){
            return 2;
        }else if(phone.substring(0, 3).equals("136")){
            return 3;
        }else if(phone.substring(0, 3).equals("137")){
            return 5;
        }else{
            return N;
        }
    }
}
```

最终所有的数据，分成了 6个分区 。确定几个细节：

1、分区的真实个数是由最大值来决定的，不是由返回值的个数来决定。
2、你在确定分区的编号的时候，一定要注意：从0开始，使用自增的整数，不要空。

```
job.setParititoner(MyPartitioner.class);
job.setNumberReduceTasks(number = N+1);
```

如果按照上述代码进行分区设置，则该 MapReduce 程序会运行 6 个 ReduceTask，编号分别是：0 1 2 3 4 5

5.5.2. 关于 MapReduce 的序列化

ZooKeeper 的序列化机制；

```
class Student implements Record{
    serialize()
    deserialize()
}
```

Hadoop 的序列化：

```
public interface Writable {

    // 序列化方法
    void write(DataOutput out) throws IOException;

    // 反序列化方法
    void readFields(DataInput in) throws IOException;
}
```

```
public interface WritableComparable<T> extends Writable, Comparable<T> {
}
```

总结两个要点：

```
1、如果你自定义的类要作为value的类型，那么该类只需要去实现Writable接口
2、如果i自定义的类要作为key的类型，那么必须实现：WritableComparable
因为：shuffle过程中，有排序的动作，所以，你自定义的类，必须要有能比较大小的能力
```

5.5.3. 关于 MapReduce 的自定义分组

某些需求可能会出现：分组规则 和 排序规则 不一致。

来看一个需求便懂：

```
求出每门课程参考学生成绩最高的学生的信息：course，name 和 score
```

数据格式：

```
course,name,score
course,name,score
course,name,score
course,name,score
course,name,score
....
```

两个细节：

```
1、shuffle中的排序：course,score
    if(o1.getCourse() == o2.getCourse()){
        return o1.getScore() - o2.getScore();
    }else{
        return o1.getCourse() - o2.getCourse();
    }
2、分组规则：course
    if(o1.getCourse() == o2.getCourse()){
        return 0
    }else{
        return 1;
    }
```

reduce 方法中，同样能拿到这一组的所有数据！(数据量大量小的问题不确定！)

则对于 MapReduce 应用程序来说，需要指定两个规则。

```
public class MyWritableComparator implements RawComparator, Configurable {

    // ....

    // 排序顺序：最终分组依据是该方法，如果在比较两个元素的时候，返回0，则表示：该两个元素的值相等
    public int compare(WritableComparable a, WritableComparable b) {
        return a.compareTo(b);
    }

    // ....
}
```

```
class DataKey implements WritableComparable<DataKey>{

    private String course;
    private String name;

    @Override
    public void write(DataOutput out) throws IOException {
        out.writeUTF(course);
        out.writeUTF(name);
    }

    @Override
    public void readFields(DataInput in) throws IOException {
        this.course = in.readUTF();
        this.name = in.readUTF();
    }

    // 排序顺序：该方法决定数据的先后顺序，如果没有通过
    // job.setGroupingComparatorClass(MyWritableComparator.class);
    // 方法设置额外的排序规则的话，该方法即是排序规则，也是分组规则。
    // 返回的大小关系，用来决定数据的先后顺序关系
    // 返回的是否等于0的结果，用来决定这两条数据是否是同一组的数据
    @Override
    public int compareTo(CourseScore o) {
        return o.getCourse().compareTo(this.getCourse());
    }
}
```

5.6. MapReduce Join实现

在分布式计算引擎中，实现 Join 的思路有两种：

1、MapJoin，顾名思义，Join 逻辑的完成是在 Mapper 阶段就完成了，这是假定执行的是 MapReduce 任务，如果是 Spark 任务，表示只用一个 Stage 就执行完了 Join 操作。

```
优点：避免了两阶段之间的shuffle，效率高，没有shuffle也就没有了倾斜。
缺点：多使用内存资源，只适合大小表做join的场景
```

2、ReduceJoin，顾名思义，Join 逻辑的完成是在 Reducer 阶段完成的。那么如果是 MapReduce 任务，则表示 Mapper 阶段执行完之后把数据 Shuffle 到 Reducer 阶段来执行 Join 逻辑，那么就可能导致数据倾斜。如果是 Spark 任务，意味着，上一个 stage 的执行结果数据 shuffle 到 下一个 stage 中来完成 Join 操作，同样也可能产生数据倾斜。

```
优点：这是一种通用的join，在不产生数据倾斜的情况下，能完成各种类型的join
缺点：会发生数据倾斜的情况
```

三个要素：

```
1、按照链接条件做key去扫描 两个表的数据，用查询字段当做value
2、在value中添加 flag，用来标识这个value来自于a表还是来自于b表
3、在reducer中，根据标识来把一组key相同的values进行标识区分，区分完之后的两个集合做笛卡尔积。
```

5.6.1. MapJoin 实现

见 Hive 调优！

5.6.2. ReduceJoin 实现

见 Hive 调优！

6. 本次课程总结

本次课程涉及到的内容其实很多，总结来说，两个大的方面：

```
1、MapReduce 的架构设计和工作原理
    MapReduce架构设计
    MapReduce程序编写规范总结
    MapReduce的核心Shuffle流程
    MapReduce并行度决定机制
2、MapReduce 企业最佳实践
    MapReduce序列化和分区分组
    MapReduce Join实现
```

首先，第一部分内容告诉你：MapReduce 是在什么样的背景下产生的，并且它的工作机制是什么样的。

然后，第二部分内容告诉你：你在使用 MapReduce 的时候，需要注意的一些细节。

要学好 MapReduce，或者说，真正理解 MapReduce 的设计精髓，最重要的就是理解两个问题：

```
1、为什么 MapReducer 有 Reducer 阶段，就一定会有 排序的动作？
2、为什么 MapReduce 的数据处理模型是 Key-Value 类型？
```

7. 本次课程作业

数字排序并加序号源数据：

第一个数据文件：data-source-1.data

```
2
32
654
32
15
```

第一个数据文件：data-source-2.data

```
756
65223
5956
22
650
92
26
54
6
```

2、最终结果：

第一个结果文件：result-0

```
1   2
2   6
3   15
4   22
5   26
```

第二个结果文件：result-1

```
6   32
7   32
8   54
9   92
10 650
```

第三个结果文件：result-2

```
11 654
12 756
13 5956
14 65223
```

3、作业要求

```
1、不能在本地运行得到结果，必须在集群运行得到结果
2、必须不能使用一个 ReduceTask 去执行，必须使用多个 ReduceTask 来执行。
```